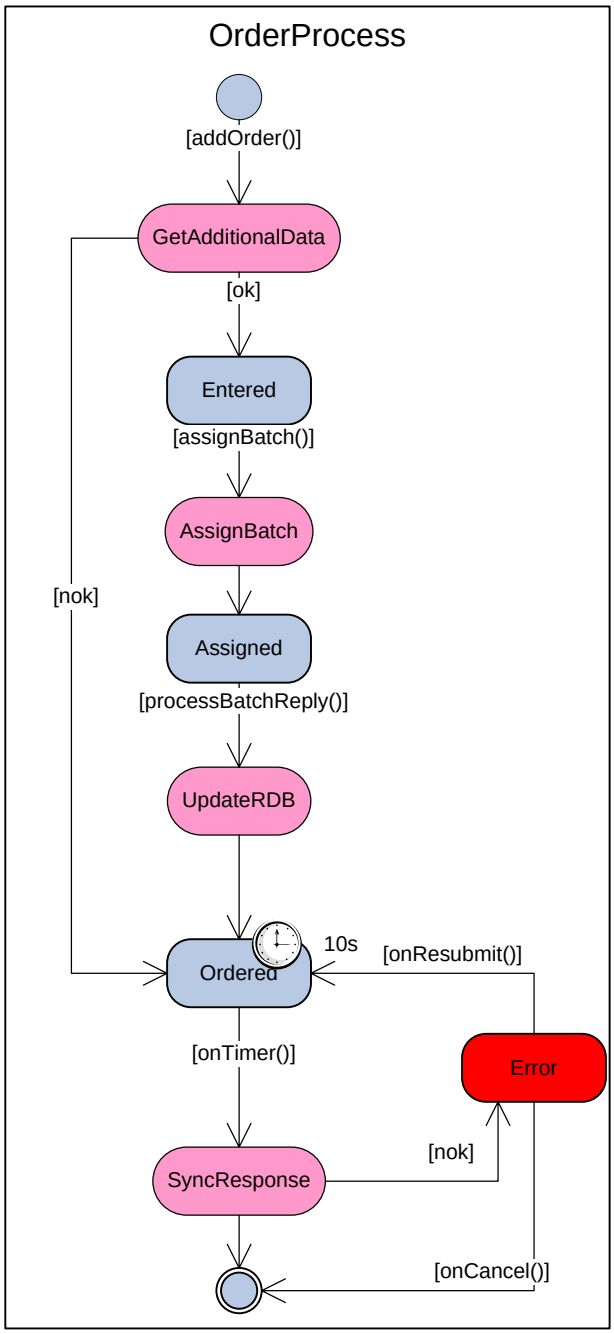
1 Process Framework

1.1 Process Definition

The OrderProcess below shows an example process definition. Process definition's are "drawn" in Visio. There is no graphical tool provided in the Process Engine framework to graphically define process definitions.



The graphical constructs which can be used in Process Engine process definitions are "State", "Action" and "Transition". There is no provision for "advanced" constructs like "fork" , "join" or "nesting". If only these are used to define the process graphically, then the definition remains implementable with the Process Engine framework!

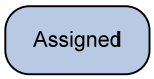
1.1.1 States

Blue boxes are States – or “wait states”. Process instances are persisted at wait states and the reception of Events trigger the transition away from the wait state. Every state must have a name. A Timer can be associated with a state.

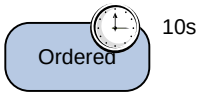
The initial state of a process is represented by:

The final state of a process is represented by:

A state without any associated Timer:



A state with an associated Timer:



A state has an **onEntry**() method which is called each time the state is entered at the end of a transition. This includes the initial state of a Process. This onEntry method can be used to conveniently set a process timer.

A state has an **onExit**() method which is called each time a transition is taken from the State. The onExit() method is not called on a final state.

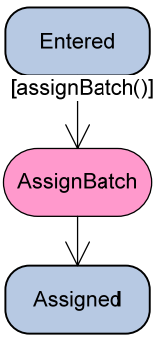
1.1.2 Actions

Pink boxes are actions. Actions are functionalities which are performed during the transition.

Actions simply represent functionality which is performed during a transition.

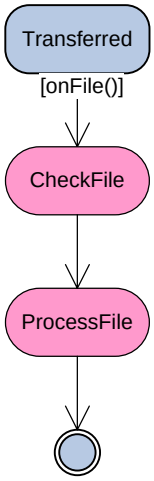
1.1.3 Transitions

A Transition represents the processing of an Event or a Timer starting at a State and ending at another (possibly same State), through Actions.



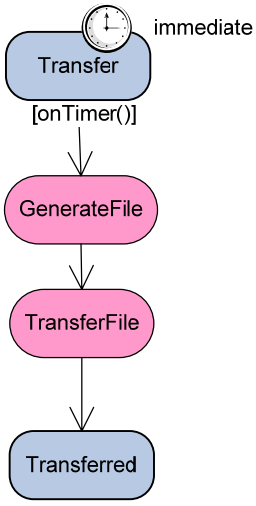
Every transition start is guarded by a “condition” – above [assignBatch()]. This means that the assignBatch() “condition” triggers the transition. The determination of the condition is some function of the Event[1] data.

The transition can flow through any number of action states like:



Where the „CheckFile“ and „ProcessFile“ functionalities are performed in the transition from the „Transferred“ state to the final state.

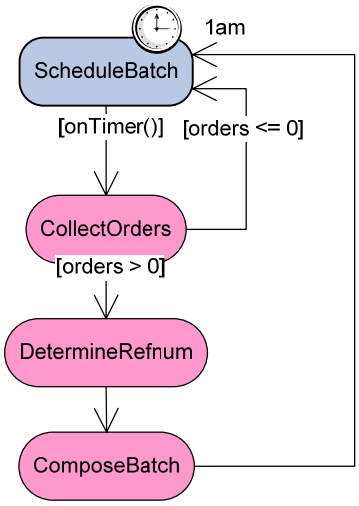
Timers can trigger transitions – they can be seen as a kind of Event – but they are not represented as Events in this framework.



The above transtion represents a timer triggering taking a ProcessInstance from a “Transfer” state to a “Transferred” state by performing the “GenerateFile” and “TransferFile” actions.

According to this framework the following is a single Transition:

[1] A more general process engine makes the current process instance state also determine the condition.

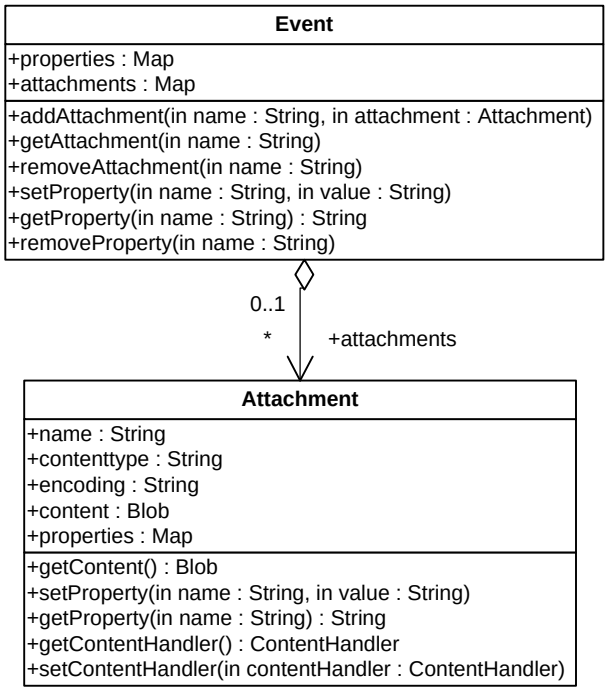


Above, a Timer event triggers a self transition. The execution of the Transition can take multiple “routes”, for instance “DetermineRefnum” and “ComposeBatch” are two actions performed if the logical condition “orders > 0” is met.

1.2 Event Model

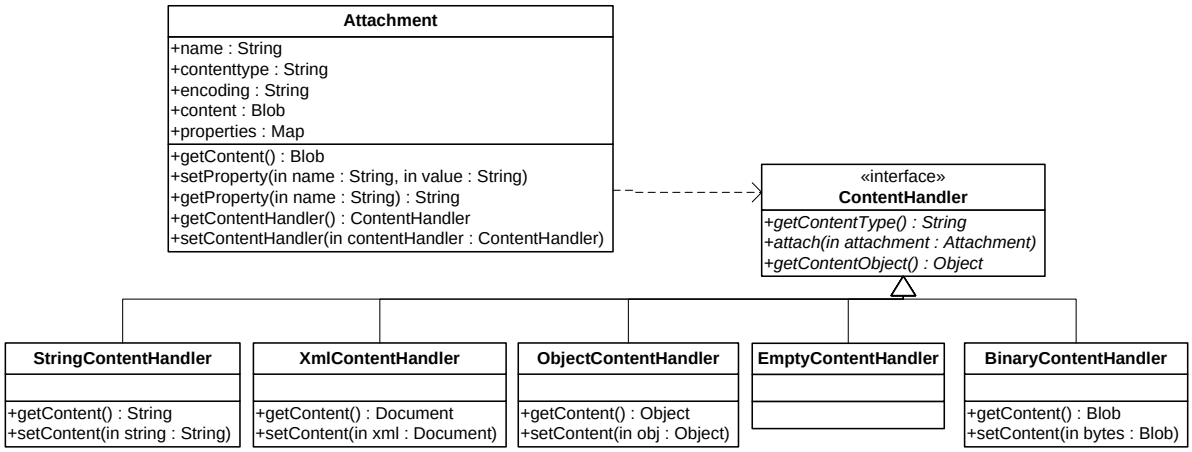
1.2.1 Event

The Event represents a generic container for passing data between communicating entities (modules, applications or components).



An Event has Properties and any number of Attachments.

For handling convenience (and borrowing from the “Email” data structure), the Attachments are extended with “ContentHandler” concept.



Content can be linked to attachments through ContentHandlers, there is one for the most used data types in java applications.

1.2.1.1 Constructing an Event

The following code fragment creates an ”Object” Event and addresses this to the ”urn://OrderProcess”.

```
Event event = new Event();
Attachment reqAttachment = new Attachment( new ObjectContentHandler(createOrderRequest));
event.addAttachment( StandardEventProperties.REQUEST_ATTACHMENT_NAME, reqAttachment);
event.addProperty( StandardEventProperties.SEND_TO, "urn://OrderProcess");
```

The following fragment constructs an ”XML” Event and addresses this to the ”urn://BatchFileProcess”

```
Event event = new Event();
Attachment reqAttachment = new Attachment( new XmlContentHandler(orderBatchDocument));
event.addAttachment( StandardEventProperties.REQUEST_ATTACHMENT_NAME, reqAttachment);
event.addProperty( StandardEventProperties.SEND_TO, "urn://BatchFileProcess");
```

1.2.1.2 Using an Event

The following code fragment gets the contents of an attachment as a byte[].

```
byte[] attachmentContents =
getEvent().getAttachment(StandardEventProperties.REQUEST_ATTACHMENT_NAME).getContent();
```

If you have apriori knowledge of what the Event contains, the convenience method can be used to get at content handler data directly,ie. Here the caller knows that the content is a FileInputHandle Object

```
FileInputHandle fih =
(FileInputHandle)event.getAttachment(StandardEventProperties.REQUEST_ATTACHMENT_NAME).getContentHandlerObject();
```

Attachments can be added to Events.

```
Attachment attachment = new Attachment();
attachment.addProperty( PROPERTY_OID, oid);
attachment.addProperty( PROPERTY_METHOD, method);
```

```
    event.addAttachment(getAttachmentName(), attachment);
```

1.2.2 EventService

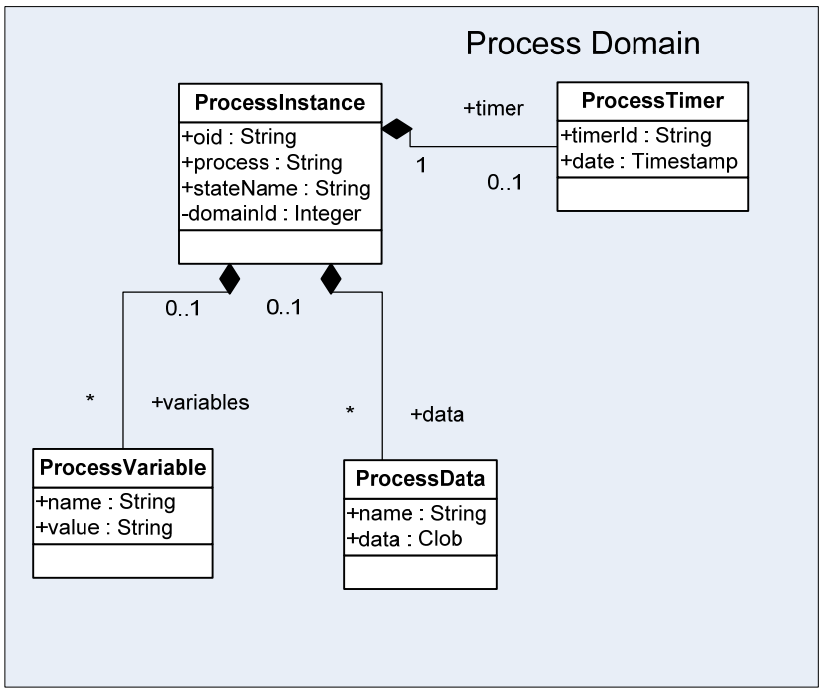
The EventService can be used for serializing and deserializing whole Events for transfer “over-the-wire”


```
<!-- Used for Event serialization / deserialization -->
<bean id="EventService" class="sunrise.itcrpe.common.service.event.EventServiceImpl">
  <property name="timestampFlag"><value>true</value></property>
</bean>
```

All known ContentHandlers must be registered with the event service implementation.

```
public EventServiceImpl() {
  contentHandlerClassMap.put( new BinaryContentHandler().getContentType(), BinaryContentHandler.class);
  contentHandlerClassMap.put( new StringContentHandler().getContentType(), StringContentHandler.class);
  contentHandlerClassMap.put( new XmlContentHandler().getContentType(), XmlContentHandler.class );
  contentHandlerClassMap.put( new ObjectContentHandler().getContentType(), ObjectContentHandler.class);
  contentHandlerClassMap.put( new EmptyContentHandler().getContentType(), EmptyContentHandler.class);
}
```

1.3 Domain Model



1.3.1 ProcessIntance Entity

The ProcessInstance domain entity can represent a single process instance for **ANY** Process. A ProcessInstance entity has:

- A set process variables (name – value pairs)
- A set of process data (name – large value pairs)
- A state (different values depending on the Process).
- A domainId – convenience attribute to link a process to an Entity of the domain model. This is used in IpAccess module to link the processes to the IpAccessAccount entities.
- An optional Timer – popping at a given absolute time.

1.3.2 ProcessInstance DAO

The DAO for the ProcessInstance domain model enables persistency of the instances and has several “finders”.

```
public interface ProcessInstanceDAO {

  public ProcessInstance findByOid( String processName, String oid );
 public List findByState(String processName, String stateName);
 public List findByVariable(String processName, String variableName, String variableValue );
  public List findByTimerReady(String processName, int maxTimers);
 ...
}
```

The DAO can be used to find instances of ANY Process with several criteria – state, variable value, oid and ready timer.

1.3.3 Hibernate Mapping

The following hibernate mapping is used to persist the ProcessInstance domain object to the “PROCESSINSTANCE” table. The variable map is kept in table “PROCESSVARIABLE” and the data in “PROCESSDATA”. Process Timers are stored as inline components with the instance.

A unique constraint is forced on the combined “processName” and “oid” field.

Todo is indexing to facilitate timer lookup and variable search.

```
<hibernate-mapping package="sunrise.itcrpe.common.domain.process">

 <class name="ProcessInstance" table="PROCESSINSTANCE">

 <!-- Entity mapping -->
  <id name="id" column="ID" type="long">
   <generator class="native"/>
  </id>
  <version column="VERSION" name="version"/>
  <property name="creationTime" type="timestamp">
   <column name="CREATION_TIME"/>
```

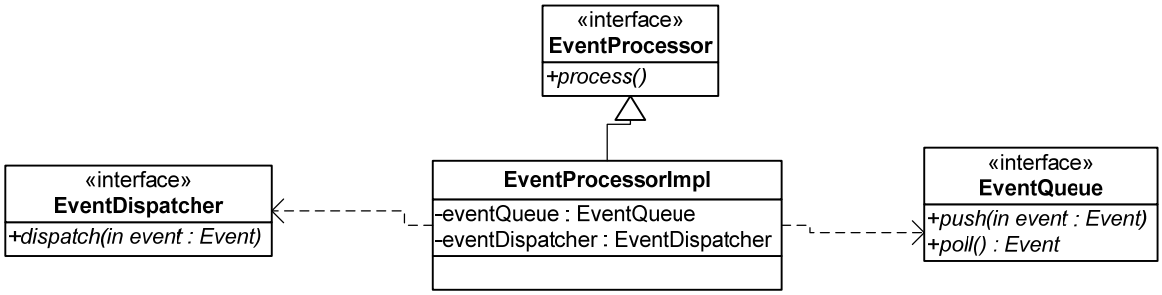
```
  </property>
  <property name="changeTime" type="timestamp">
    <column name="CHANGE_TIME"/>
  </property>

 <!-- Attribute Mapping -->
  <property name="processName">
    <column name="PROCESS_NAME" length="100" unique-key="processOid"/>
  </property>
  <property name="oid">
    <column name="OBJECT_ID" length="100" unique-key="processOid"/>
  </property>
  <property name="stateName">
    <column name="STATE_NAME" length="100"/>
  </property>

 <!-- map of variables -->
 <map name="variables" table="PROCESSVARIABLE" lazy="true">
    <key column="INSTANCE" not-null="true"/>

    <map-key column="VAR_KEY"  type="string" length="100"/>
   <element column="VAR_VALUE" type="string" length="256"/>
 </map>

 <!-- timer component -->
 <component name="timer" class="sunrise.itcrpe.common.domain.process.ProcessTimer">
     <property name="date" type="timestamp">
       <column name="TIMERDATE"/>
     </property>
     <property name="timerId">
       <column name="TIMERID" length="64"/>
     </property>
  </component>

  <!-- map of data -->
  <map name="data" table="PROCESSDATA" lazy="true">
    <key column="INSTANCE" not-null="true"/>

    <map-key column="VAR_KEY"  type="string" length="100"/>
   <element column="VAR_DATA" type="binary" length="10000"/>
  </map>

 </class>

</hibernate-mapping>
```

Currently it is not forseen to have different process table for different XIP modules.

1.4 Process Framework Components

1.4.1 EventProcessor

The EventProcessor processes queued Events by dispatching them via the EventDispatcher.



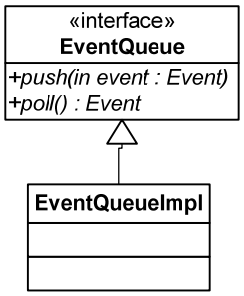
The spring configuration is:

```
<bean id="EventProcessorTarget" class="sunrise.itcrpe.common.service.event.EventProcessorImpl">
  <property name="eventQueue"><ref local="EventQueue"/></property>
  <property name="eventDispatcher"><ref local="EventDispatcher"/></property>
</bean>
<bean id="EventProcessor" class="org.springframework.tx.interceptor.TrProxyFactoryBean">
  <property name="transactionManager"><ref bean="poc.dao.TransactionManager"/></property>
  <property name="target"><ref local="EventProcessorTarget"/></property>
  <property name="transactionAttributes">
    <props>
      <!-- all events processed at one time are in the same transaction -->
      <prop key="process">PROPAGATION_REQUIRED</prop>
    </props>
  </property>
</bean>
```

The process() method requires a transaction, so that every event processed is in the **same** transaction.

1.4.2 EventQueue

The EventQueue is a FIFO container for events. Any Event pushed into the Queue will be dispatched by the EventProcessor.

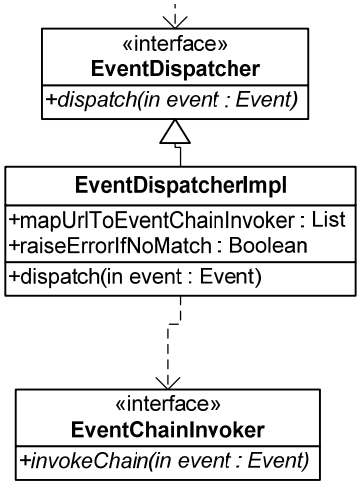


The implementation provides a queue **per Thread.** This guarrantees that all events processed by a single thread share the same transaction.

```
<bean id="EventQueue" class="sunrise.itcrpe.common.service.event.EventQueueImpl">
</bean>
```

1.4.3 EventDispatcher

The EventDispatcher dispatches single Events to an EventChainInvoker depending on the Event's "StandardEventProperties#SEND_TO" property.



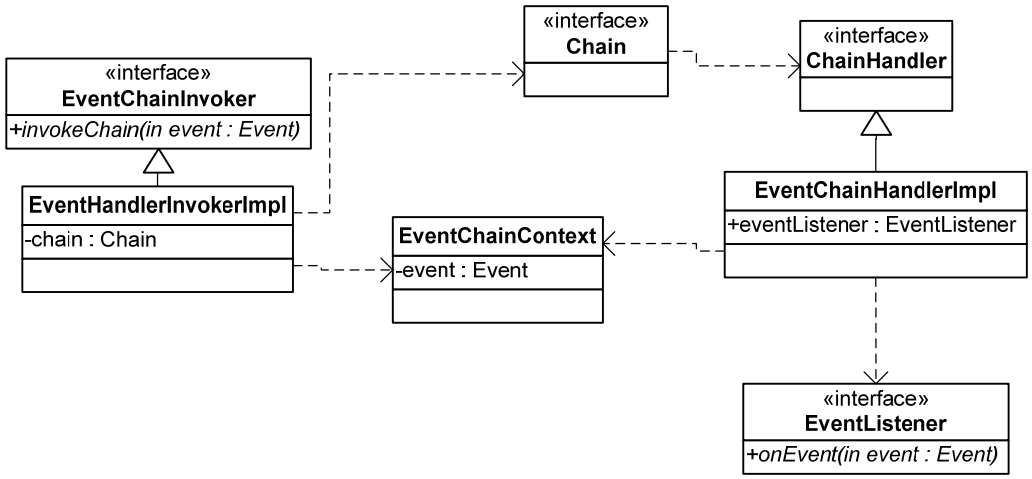
The spring configuration example:

```
<bean id="EventDispatcher" class="sunrise.itcrpe.common.service.event.EventDispatcherImpl">
  <property name="noMatchCausesErrorFlag"><value>true</value></property>
  <property name="dispatchMap">
    <map>
      <!-- BatchSchedulerProcess does not receive any events -->
      <entry>
        <key><value>urn://OrderProcess</value></key>
        <ref local="OrderProcessHandlerChainInvoker"/>
      </entry>
      <entry>
        <key><value>urn://BatchFileProcess</value></key>
        <ref local="BatchFileProcessHandlerChainInvoker"/>
      </entry>
    </map>
  </property>
</bean>
```

The urn://OrderProcess events are dispatched to "OrderProcessHandlerChainInvoker" bean. If noMatchCausesErrorFlag is true, then there will be a runtime error generated if an event is dispatched which does not match any URL.

1.4.4 EventChainInvoker and EventChainHandler

The EventChainInvoker dispatches Events down the common "Chain" implementation. The EventChainHandlerImpl is a ChainHandler which works together with the Invoker to invoke an EventListener.



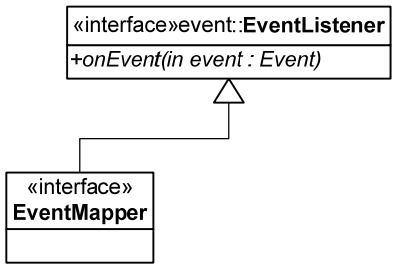
With this construct, chains of EventListeners can be formed.

```
 <bean id="OrderProcessHandlerChainInvoker"
class="sunrise.itcrpe.common.service.event.chain.EventChainInvokerImpl">
   <property name="chain">
     <ref bean="OrderProcessHandlerChain"/>
   </property>
 </bean>

 <bean id="OrderProcessHandlerChain" class="sunrise.itcrpe.common.chain.ChainFactoryBean">
   <property name="handler">
     <list>
       <bean class="sunrise.itcrpe.common.service.event.chain.EventChainHandlerImpl">
         <property name="listener"><ref bean="poc.service.OrderProcessEventMapper"/></property>
        </bean>
       <bean class="sunrise.itcrpe.common.service.event.chain.EventChainHandlerImpl">
         <property name="listener"><ref bean="poc.service.OrderProcess"/></property>
        </bean>
     </list>
   </property>
 </bean>
```

Currently defined EventListeners are EventMapper and Process.

1.4.5 EventMapper

EventMappers are EventListeners which are plugged into EventHandlerChains and help Processes determine the “oid” of ProcessInstances and “method” names based on incoming Event data.



```
<bean id="poc.service.OrderProcessEventMapper" class="sunrise.poc.process.orderprocess.OrderProcessEventMapper">
  <property name="attachmentName"><value>OrderProcessEventMapContext</value></property>
</bean>
```

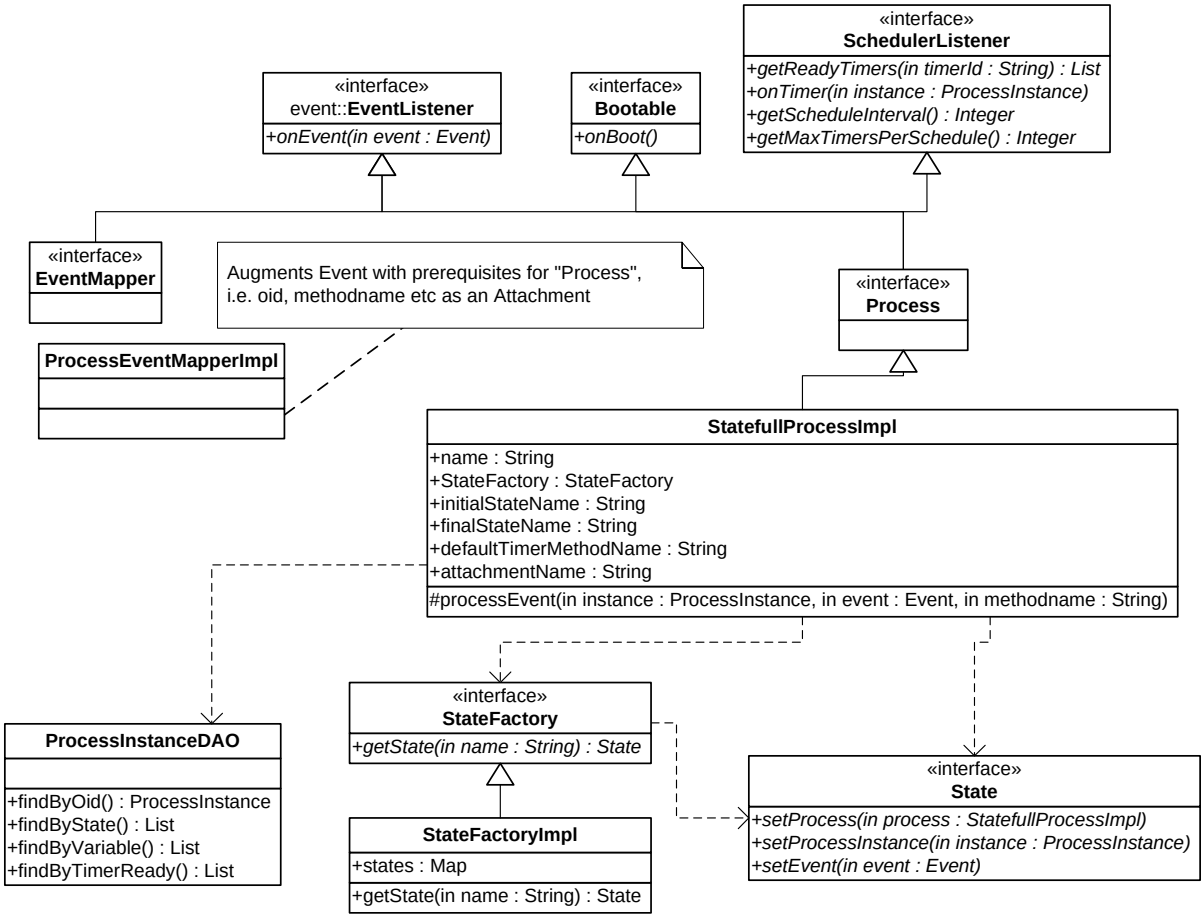
The OrderProcessEventMapper analyzes incoming Event data and sets the oid and method if it recognizes the data.

```
public class OrderProcessEventMapper extends BaseEventMapper {

  public void onEvent(Event event) {
    Attachment requestAttachment = event.getAttachment(StandardEventProperties.REQUEST_ATTACHMENT_NAME);
    if ( requestAttachment == null ) {
      return; // not interrested if it doesnt contain the data i'm looking for.
    }
    // determine the oid and method from the incoming data
    Object requestObject = requestAttachment.getContentHandlerObject();

    String oid = null;
    String method = null;

    if ( requestObject instanceof CreateOrderRequest ) {
      CreateOrderRequest req = (CreateOrderRequest)requestObject;

      oid = req.getOrder().getOrderId();
      method = "addOrder";
    } ...

    if ( oid != null && method != null ) {
      // use standard properties for oid and method communiction to the Process
      Attachment attachment = new Attachment();
      attachment.addProperty( PROPERTY_OID, oid);
      attachment.addProperty( PROPERTY_METHOD, method);

      // add the attachment for ExampleProcess onto the Event
      event.addAttachment(getAttachmentName(), attachment);
    }
  }
```

Note that the attachmentName is also configured on the OrderProcess – so that the event mapper and Process have a common understanding of the attachment to use for handling the Event.

1.4.6 Process

The main component which enables statefull Process execution. A Process is a “Bootable” “EventListener” which can be linked to “Scheduler”.



A Process receives events via the EventChainInvoker. Typically it is placed in the EventChain after it's corresponding EventMapper. So the EventMapper will analyze events for the Process before the Event reaches the Process.

The StatefullProcessImpl provides a implementation which works as follows:

- For each Event received in onEvent which has the named Attachment (set by EventMapper), the "oid" and "method" name are retrieved from the Attachment.
- If there doesnt exists already a ProcessInstance for this Process with the oid – then one is created in the configured initial state (determination and instantiation through ProcessInstanceDAO).
- The State for the state is retrieved by the StateFactory given it's name.
- The State is "primed" by setting the Event, Process and ProcessInstance on the State.
- The onExit() method is called on the current State.
- The method "method" is called on the State. (This may change the instance's State).
- The onEnter() method is called on the new State.
- After the method "method" returns, if the ProcessInstance is in the final state, then the process instance is deleted. (future add archiving).

The functionality is similar for "Timer" events which are triggered by the Scheduler.

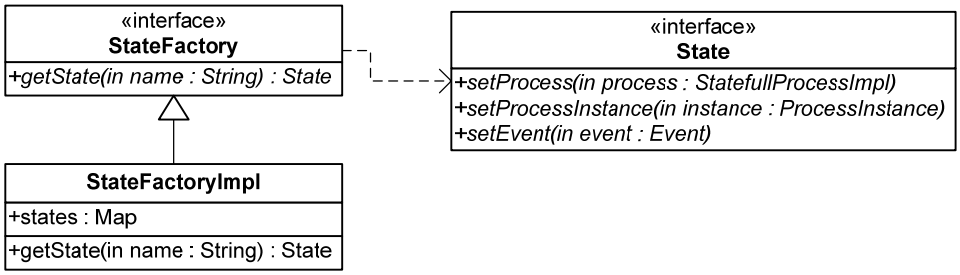
States are non singleton Spring beans which can have references to any other Spring bean, for example the EventQueue if any new events are created and should be processed in the same transaction as the current event.

```
<bean id="poc.service.OrderProcessTarget" class="sunrise.itcrpe.common.service.process.StatefullProcessImpl">
  <property name="name"><value>OrderProcess</value></property>
  <property name="attachmentName"><value>OrderProcessEventMapContext</value></property>
  <property name="eventProcessor" ref="EventProcessor"/>
  <property name="maxTimersPerSchedule"><value>10</value></property>
  <property name="stateFactory" ref="poc.service.OrderProcessStateFactory"/>
  <property name="processInstanceDAO" ref="poc.dao.ProcessInstanceDAO"/>
</bean>
<bean id="poc.service.OrderProcess" class="org.spr....TransactionProxyFactoryBean">
  <property name="transactionManager"><ref bean="poc.dao.TransactionManager"/></property>
  <property name="target"><ref local="poc.service.OrderProcessTarget"/></property>
  <property name="transactionAttributes">
    <props>
      <!-- each event processed just joins in the current Tx -->
      <prop key="onEvent">PROPAGATION_REQUIRED</prop>
      <prop key="onBoot">PROPAGATION_REQUIRED</prop>
      <!-- each process instance timer pop requires a new Tx -->
      <prop key="onTimer">PROPAGATION_REQUIRES_NEW</prop>
      <!-- if there is a transaction setup, who cares -->
      <prop key="getReadyTimers">PROPAGATION_SUPPORTS</prop>
    </props>
  </property>
</bean>
```

If a Process needs to perform startup actions – for example creating one ProcessInstance which triggers regularly on a Timer (see poc module BatchSchedulerProcess), it needs to be hooked up to the BootService in the spring configuration. Then the boot() method will be called in a transactional context on application startup.

1.4.7 StateFactory

The Process implementations rely on StateFactories to provide States given (persisted) state names.




```
<bean id="poc.service.OrderProcessStateFactory" class="sunrise.itcrpe.common.service.process.state.StateFactoryImpl">
  <property name="stateRefs">
    <list>
      <value>poc.service.OrderProcess.InitialState</value>
      <value>poc.service.OrderProcess.EnteredState</value>
      <value>poc.service.OrderProcess.AssignedState</value>
      <value>poc.service.OrderProcess.OrderedState</value>
      <value>poc.service.OrderProcess.FinalState</value>
```

```
    </list>
  </property>
  <property name="initialStateRef"><value>poc.service.OrderProcess.InitialState</value></property>
  <property name="finalStateRef"><value>poc.service.OrderProcess.FinalState</value></property>
  <property name="timerMethodName"><value>onTimer</value></property>
</bean>
```

The stateRefs is a list of Bean “names” which reference spring beans representing individual states.

1.4.8 States

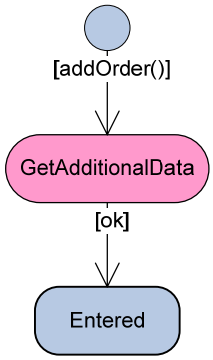
The individual states are declared (non singletons) in the Spring configuration which are linked to StateFactories which are linked to Processes.

Each State contains the code for the Process “Actions” as defined in the Process Definition.

```
 <bean id="poc.service.OrderProcess.InitialState" class="sunrise.poc.process.orderprocess.state.InitialState"
singleton="false">
 </bean>

 <bean id="poc.service.OrderProcess.EnteredState" class="sunrise.poc.process.orderprocess.state.EnteredState"
singleton="false">
 </bean>

 <bean id="poc.service.OrderProcess.AssignedState" class="sunrise.poc.process.orderprocess.state.AssignedState"
singleton="false">
   <property name="orderedTimerInterval"><value>10000</value></property>
 </bean>

 <bean id="poc.service.OrderProcess.OrderedState" class="sunrise.poc.process.orderprocess.state.OrderedState"
singleton="false">
 </bean>

 <bean id="poc.service.OrderProcess.FinalState" class="sunrise.poc.process.orderprocess.state.FinalState"
singleton="false">
 </bean>
```

A convenient base class is provided which has getters and setters for the Event, Process and ProcessInstance properties.

For example the following code example represents this process fragment




```
public class InitialState extends BaseState {

  protected static Log log = LogFactory.getLog(InitialState.class);

  public String getName() {
    return "InitialState";
  }

  public void addBatch() {
    throw new UnsupportedOperationException("addBatch");
  }

  public void updateOrder() {
    throw new UnsupportedOperationException("updateOrder");
  }

  public void addOrder() {
    log.info("addOrder");

    CreateOrderRequest createOrderRequest = (CreateOrderRequest)getEvent().getAttachment(
StandardEventProperties.REQUEST_ATTACHMENT_NAME ).getContentHandlerObject();

    // the order's orderId has become the oid of the process instance through the request mapper
    // we take the order's requestdata and persist this as a process variable

    getProcessInstance().addVariable("requestdata", createOrderRequest.getOrder().getRequestdata());

    CreateOrderResponse createOrderResponse = new CreateOrderResponse();
    createOrderResponse.setOrderId(getProcessInstance().getOid());

    // give a sync response back to the ws layer, by attaching it to the event
    Attachment resAttachment = new Attachment( new ObjectContentHandler(createOrderResponse));
    getEvent().addAttachment( StandardEventProperties.RESPONSE_ATTACHMENT_NAME, resAttachment);

    // transition to EnteredState
    getProcessInstance().setStateName( new EnteredState().getName() );
  }
}
```

The addOrder method fetches the CreateorderRequest event payload and processes it.

A synchronous reply is piggybacked on the Event back to the Event dispatcher.

```
Attachment resAttachment = new Attachment( new ObjectContentHandler(createOrderResponse));
getEvent().addAttachment( StandardEventProperties.RESPONSE_ATTACHMENT_NAME, resAttachment);
```

and a transition is made to the EnteredState with:

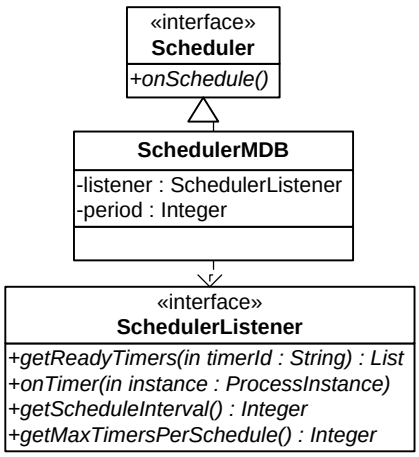
```
// transition to EnteredState
getProcessInstance().setStateName( new EnteredState().getName() );
```

If a timer needs to be scheduled, for example 60s in the future, then this must be coded by:

```
getProcessInstance().setTimer( new ProcessTimer( System.currentTimeMillis() + 60000 ) );
```

1.4.9 Scheduler

The SchedulerMDB is a MessageDrivenBean. The SchedulerMDB is configured with a ‘period’ timer interval which triggers a thread to call the configured ‘schedulerListener’.



Since EJB’s are not spring components, they are linked to the spring context with their own configuration.

In ejb-jar.xml there is a EJB descrptor for the SchedulerMDB. You only need one SchedulerMDB per Process.

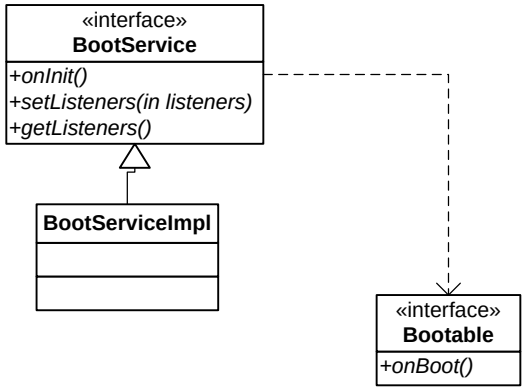
```
<message-driven>
  <ejb-name>BatchSchedulerProcessSchedulerMDB</ejb-name>
  <ejb-class>sunrise.poc.process.scheduler.SchedulerMDB</ejb-class>
  <messaging-type>sunrise.ra.scheduler.inbound.Scheduler</messaging-type>
  <transaction-type>Bean</transaction-type>
  <activation-config>
    <activation-config-property>
      <activation-config-property-name>period</activation-config-property-name>
      <activation-config-property-value>60000</activation-config-property-value>
    </activation-config-property>
    <activation-config-property>
      <activation-config-property-name>schedulerName</activation-config-property-name>
      <activation-config-property-value>poc.service.BatchSchedulerProcess</activation-config-property-value>
```

```
    </activation-config-property>
  </activation-config>
  <env-entry>
     <description>SchedulerListener bean reference name</description>
     <env-entry-name>SchedulerListener</env-entry-name>
     <env-entry-type>java.lang.String</env-entry-type>
     <env-entry-value>poc.service.BatchSchedulerProcess</env-entry-value>
  </env-entry>
  <resource-ref>
    ...
   </resource-ref>
</message-driven>
```

The environment reference is used to give the SchedulerMDB the name of the spring bean to use as the schedulerListener.

1.4.10 BootService

The BootService is used to bootstrap Processes – to enable them to do transactional setup. A Process is a Bootable entity, which can be hooked up to the BootService. See 3.5.4Ejb Context for an example spring configuration.



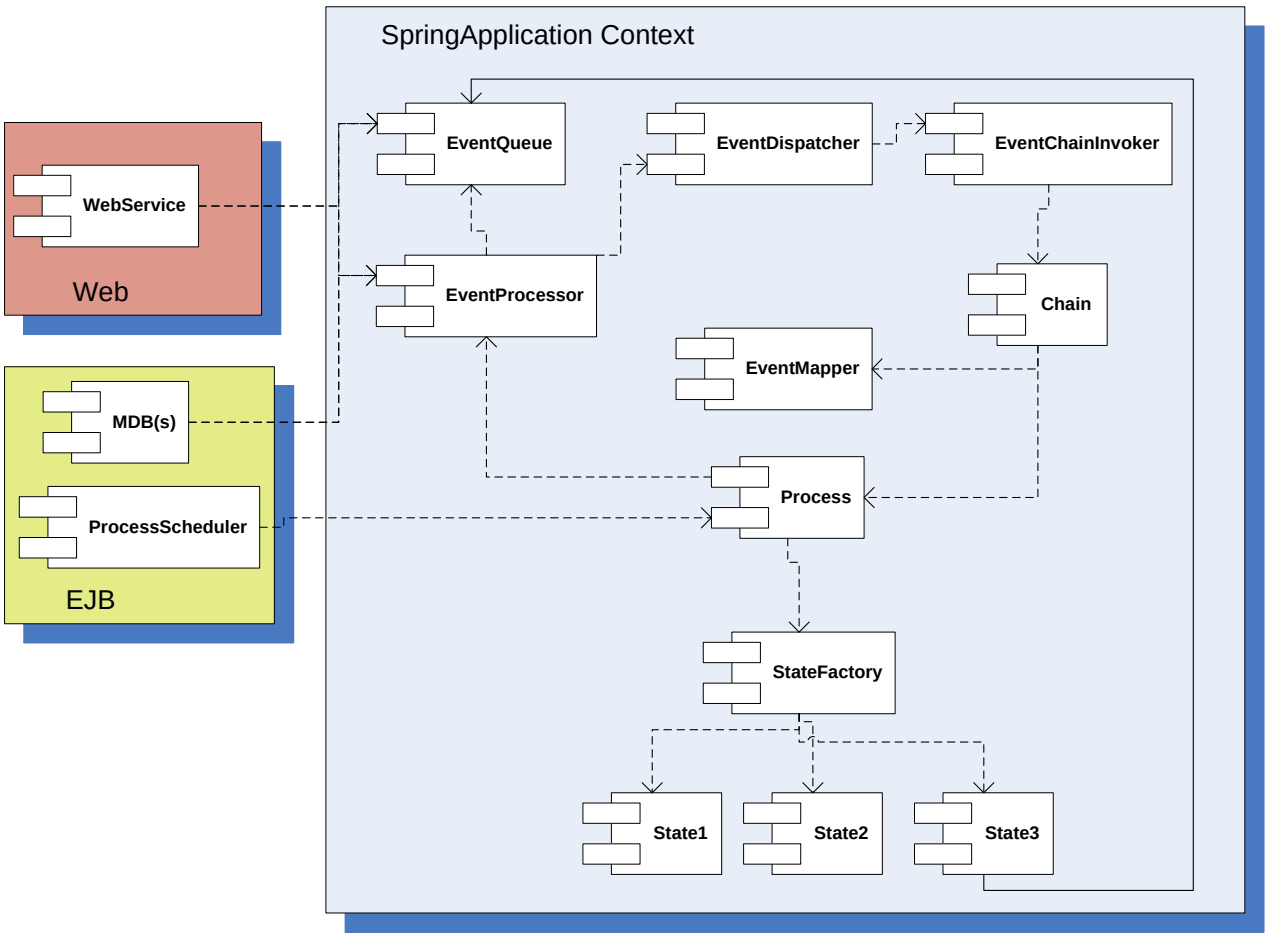
The following shows the bootstrap of the BatchSchedulerProcessImpl which requires only one ProcessInstance, which is timer triggered. The onBoot() method instantiates a new ProcessInstance with a fixed “oid” if not already existing.

```
public class BatchSchedulerProcessImpl extends StatefullProcessImpl {

  public static final String BatchSchedulerSingletonOid = "BatchScheduler";

   protected static Log log = LogFactory.getLog(BatchSchedulerProcessImpl.class);

  /**
   * Make a single instance of the BatchScheduler with process oid
   * 'BatchScheduler'
   */
  public void onBoot() {
    log.info("onBoot called on Process " + getName());
```

```
    String oid = BatchSchedulerSingletonOid;
    ProcessInstance processInstance = getProcessInstanceDAO().findByOid( getName(), oid );

    if ( processInstance == null ) {
      log.info("onBoot ProcessInstance " + oid + " not found - creating...");
      ScheduleBatchState state = new ScheduleBatchState();

      processInstance = new ProcessInstance();
      processInstance.setOid(oid);
      processInstance.setProcessName(getName());
      processInstance.setStateName( state.getName() );
      processInstance.setTimer(new ProcessTimer(state.getNextBatchTimerTime()));

      // it is extremely unlikely that two instances boot at exactly
      // the same time, so the possibility of concurrency failure in the
      // save below is negligible
      getProcessInstanceDAO().save(processInstance);
      log.info("onBoot ProcessInstance " + oid + " created.");
    } else {
      log.info("onBoot found " + processInstance );
    }
    log.info("onBoot finished for Process " + getName());
  }
}
```

1.5 Spring Application Context

This Diagram shows the entire spring application context necessary to support Processes. The yellow box represents EJB's which exist only in the EJB Container. The blue box contains Spring Application Context which is in both EJB and Web Container.



The functioning of event processing works like this:

- An event is placed on the EventQueue with a SendTo URL.
- The EventProcessor is called to process the queued Events.
- The EventProcessor takes the first Event from the EventQueue and calls the EventDispatcher.
- The EventDispatcher routes the Event to the appropriate EventChainInvoker depending on the SendTo URL of the Event.
- The EventChainInvoker invokes sequentially the EventMapper and then Process in the Chain.
- The EventMapper analyzes the Event and adds an attachment to it if it can be processed by the Process.
- The Process receives the Event from the Chain with an attachment which indicates the “oid” and “method” to call. The Process looks up or creates a ProcessInstance, and retrieves the State corresponding to the ProcessInstance from the StateFactory.
- The “method” of the State is called after the Event and ProcessInstance and Process is set on the State.
- The State “method” can cause new Events to be placed on the EventQueue which will be processed after the method is completed.
- The EventChainInvoker returns once the Event is dispatched all the way down the Chain.

- The EventProcessor dispatches the next event until there are none left.
- The thread of control goes back to the caller of the EventProcessor, ie. MDB, EJB or WS.

1.6 Deployment Architecture

1.6.1 Deployed Module

The <module>-ear.jar EAR is deployed which contains <module>-ejb.jar , <module>-web.war and any datasources required.

JCA resource adapters are deployed separately (before the <module> ear).

1.6.2 Common Spring Context

Since we have two Enterprise Containers running for a single Application module (J2EE Enterprise Application) – ie. WebContainer and EJBContainer, we have two separate Spring Application Contexts. To keep complexity down, we separate out a 'common' Spring Context which is loaded by both containers.

For example 'common-context.xml'

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>

   <import resource="classpath*:/sunrise/poc/dao/data-source.xml"/>
   <import resource="classpath*:/sunrise/poc/dao/data-access.xml"/>
   <import resource="classpath*:/sunrise/poc/process/processes.xml"/>

</beans>
```

1.6.3 WebApp Context

The web-context.xml loads the common spring context – and registers any JMX components specific to the web context.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>

  <import resource="classpath*:/sunrise/poc/common-context.xml"/>

</beans>
```

1.6.4 Ejb Context

The EJB context is configured to be loaded by all EJB's. The common context is included and support for the bootstrapping of individual Processes.

```
   <import resource="classpath*:/sunrise/poc/common-context.xml"/>
```

```
<!-- Bootstrap of other Processes ( must have init-method="onInit" ) -->
<bean id="BootService" class="sunrise.itcrpe.common.service.boot.BootServiceImpl" init-method="onInit">
  <property name="listeners">
    <list>
      <ref bean="poc.service.BatchSchedulerProcess"/>
      <ref bean="poc.service.OrderProcess"/>
    </list>
  </property>
</bean>
```

Processes can use this bootstrapping to create initial process Instances. See the BatchSchedulerProcess in the poc module demo.

Bootstrapping of Processes is not achieved simply by an init-method on the Process itself, since the boot() method requires a transaction context – i.e. to create new ProcessInstances which must come from “outside” the bean itself, so the transaction proxy wrapping the Process can establish the transactional context.

1.6.4.1 Achieving a single SpringContext for all EJB Components

A single spring configuration is shared for all EJB components (MessageDrivenBeans and StatelessSessionBeans)

```
  public void setSessionContext(SessionContext sessionContext) {
    super.setSessionContext(sessionContext);
    setBeanFactoryLocator(ContextSingletonBeanFactoryLocator.getInstance());
    setBeanFactoryLocatorKey("SingletonEjbSpringContext");
  }
```

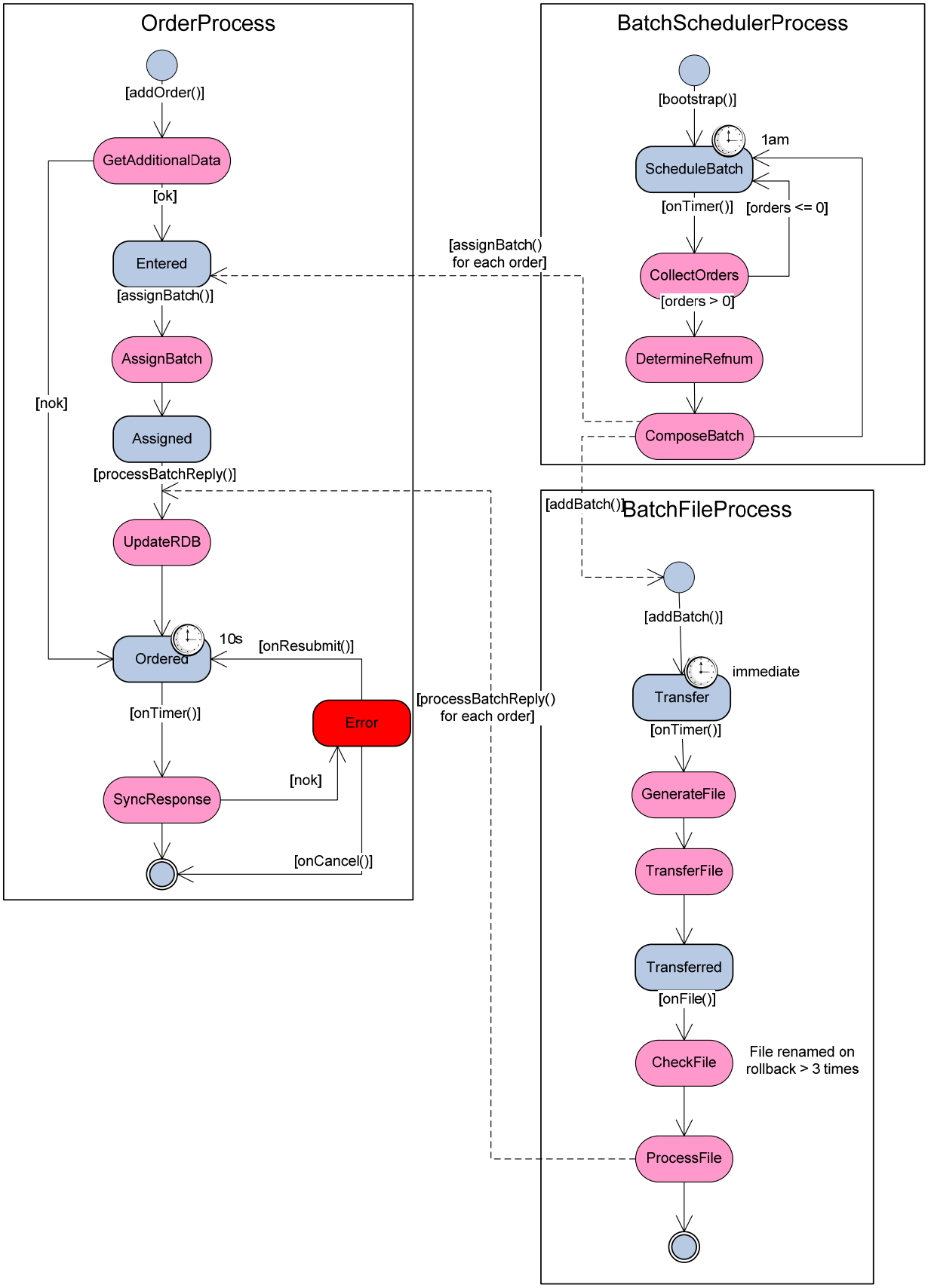
File ‘beanRefContext.xml’ must be placed in the EJB classpath (packaged with classes into ‘<module>.jar’)

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>

  <!-- the bean id 'SingletonEjbSpringContext' must never be changed -->
  <!-- it is referenced in EJB code 'setSessionContext()' methods -->
  <bean id="SingletonEjbSpringContext" class="org.springframework.context.support.ClassPathXmlApplicationContext">
    <constructor-arg>
      <list>
        <value>/sunrise/poc/ejb-context.xml</value>
      </list>
    </constructor-arg>
  </bean>
</beans>
```

Note the use of the hardcoded name ‘SingletonEjbSpringContext’

1.7 POC Module

The POC module is a proof of concept module for the XIP process framework. The following process models were implemented:


OrderProcess
[addOrder()]
GetAdditionalData
[ok]
Entered
[assignBatch()]
AssignBatch
[nok]
Assigned
[processBatchReply()]
UpdateRDB
10s
[onResubmit()]
Ordered
[onTimer()]
Error
[nok]
SyncResponse
[onCancel()]
[assignBatch()
for each order]
[processBatchReply()
for each order]
BatchSchedulerProcess
[bootstrap()]
1am
ScheduleBatch
[onTimer()]
[orders <= 0]
CollectOrders
[orders > 0]
DetermineRefnum
ComposeBatch
[addBatch()]
BatchFileProcess
[addBatch()]
immediate
Transfer
[onTimer()]
GenerateFile
TransferFile
Transferred
[onFile()]
CheckFile
File renamed on
rollback > 3 times
ProcessFile